ELSEVIER

Journal of Systems Architecture 45 (1998) 519–542

JOURNAL OF SYSTEMS ARCHITECTURE

Tools for code optimization and system evaluation of the image processing system PAPRICA-3

Massimo Bertozzi [1], Alberto Broggi [*]

Dipartimento di Ingegneria dell'Informazione, Università di Parma, I-43100 Parma, Italy

Abstract

This paper presents the complex environment that was built to ease the prototyping of real-time applications on the PAPRICA-3 massively parallel system. Applications are developed in C++ using high level data types and the corresponding Assembly code is automatically created by a code generator. A stochastic code optimizer takes the assembly code and improves it according to a genetic approach; due to the high computational power required by this approach, the stochastic code optimizer was implemented with MPI and runs in parallel on a cluster of workstations. The availability of this complex environment allowed to test the performance of the system and to tune it according to some target applications before the actual development of the hardware. For this purpose a system-level simulator was also built to determine the number of clock cycles required to run a specific segment of code. The whole environment has been used to validate possible solutions for the hardware system and to develop, test, and tune several real-time image processing applications. The hardware system is now completely defined. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Code optimization; System evaluation; Image processing system

1. Introduction

Real-time image processing applications require powerful engines. A lot of general-purpose processors nowadays can deliver sufficient computational power, but when specific requirements (such as a low power

[*] Corresponding author. E-mail: broggi@ce.unipr.it
[1] E-mail: bertozzi@ce.unipr.it

consumption, a small physical size, and a low production cost [2]) must be met, a special-purpose system with a deep match between the architecture and the application becomes mandatory.

A lot of special-purpose architectures have been conceived, designed, and implemented [9], but not always the design of new systems has been accompanied with the development of proper environments to ease the prototyping and tuning of applications. In general, complex hardware systems require either a specialized skill for its programming or highly optimizing compilers and environments: in the first case the user must be aware of all internal details of the machine to be able to exploit its sophisticated characteristics, while in the latter the user programs with high level languages, thus leaving to an optimizing compiler or code generator the hard task of writing efficient code. Obviously this complex task becomes even harder when the target of the project is to reach real-time performance on a custom system.

With the continuously growing possibilities offered by hardware today, words such as *pipeline, branch prediction, dynamic scheduling*, and *superscalar architectures* are no more restricted to a small specialists elite but are spreading also in a standard and general environment; even low-cost special-purpose processors are designed focusing on these techniques [11]. Thus, code optimization by-hand is not only difficult and time-consuming, but generally produces code with an efficiency lower than what could be achievable using automatic tools.

This work presents the programming environment that was developed to ease the prototyping of real-time applications on the PAPRICA-3 system, a special-purpose coprocessor designed to run real-time image processing tasks developed in cooperation with the Polytechnic of Turin, Italy. To exploit the natural parallelism of low-level processing of images, PAPRICA-3 is composed of a high number of processors working in SIMD fashion. Moreover, the internal structure of each single processor features a complex pipeline which allows to take advantage of the intrinsic parallelism of a generic program [24]. Thus, both *Spatial Parallelism and Instruction-Level Parallelism* [21] are used to boost performance, and must be taken into account when writing Assembly code.

The PAPRICA-3 programming environment has thus been structured to hide these two instances of parallelism. Spatial parallelism, namely the distribution of the data over the set of processing elements, is automatically handled by a high level language, which allows the user to describe the application employing abstract data types; a code generator then builds the corresponding assembly program using a set of parameterized libraries. Conversely the exploitation of instruction-level parallelism, namely the simultaneous execution of portions of each single assembly instruction, is handled by a code optimizer, which modifies the assembly code produced by the previous tool in order to minimize the number of clock cycles required to run the code.

Pipelining techniques were first addressed at least 25 years ago [20], and in this period a great variety of algorithms has been studied and implemented to improve code efficiency. Almost all of them rely on deterministic approaches, which lead to NP-complete problems [10]. Conversely in this work a stochastic approach is considered: unlikely other optimization processes, a "population" of modified versions of the original program evolves according to a genetic methodology. This approach allows to achieve higher optimization levels, but unfortunately it requires a higher computational power (in terms of both memory

[2] For example for the development of a vision-based system for the automatic driving of vehicles or robots [5].

size and CPU time). To face this requirements the optimizer has been implemented on a cluster of workstations using standard MPI (Message Passing Interface [19]) libraries.

This work is organized as follows: Section 2 briefly introduces the PAPRICA-3 system; Section 3 presents the programming environment and the C++ classes that model the new data types and introduces the code optimization tool; Sections 4 and 5 describe how the stochastic optimizer works and how performances are evaluated in absence of the real hardware; Section 6 describes the implementation of each logical part of the environment and Section 7 presents a case study and describes possible future extensions of the environment; Section 8 ends the paper with some conclusive remarks.

2. Brief overview of the PAPRICA-3 system

PAPRICA-3 [7] is a massively parallel system dedicated to run real-time image processing tasks. The system specifications are now completely defined, and the ICs are currently under fabrication; the hardware system, a PCI board that will be connected to a standard PC, will be available in early 1998.

2.1. Internal architecture

As shown in Fig. 1(a), the core of the processor is a dedicated SIMD cellular architecture based on a linear array of Q identical single-bit Processing Elements (PEs); a single chip contains 32 processors, and the architecture is modular. The array is connected to an external Image Memory via a bidirectional Q-bit data bus; therefore each memory access transfers a complete vector of Q pixels at once, 1 bit per pixel. The rationale behind this system is that the size of the Processor Array (PA) matches exactly the width of the input image, thus avoiding the PE virtualization problem, which has been proven to be a critical design issue in 2D arrays [8]. The PEs contain a fairly complex pipeline [11]; since the PE clock cycle is twice as fast

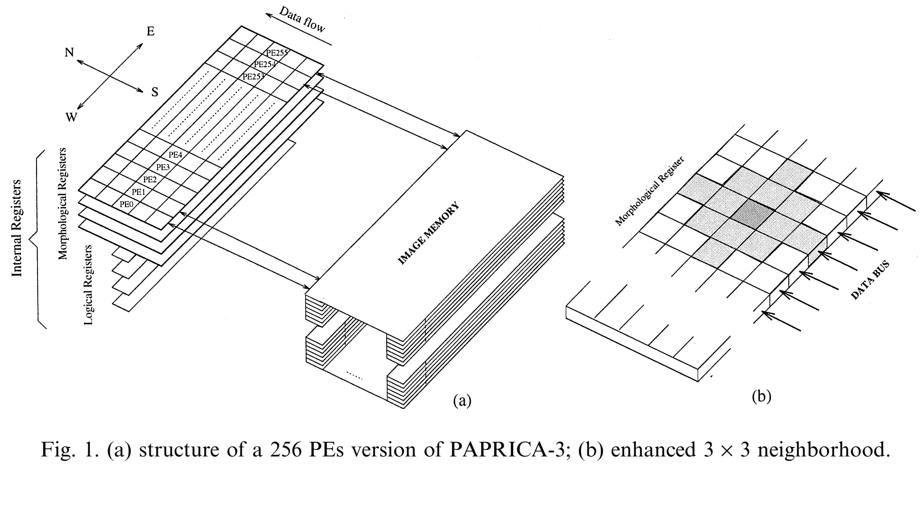

Fig. 1. (a) structure of a 256 PEs version of PAPRICA-3; (b) enhanced 3 × 3 neighborhood.

as a memory access, the instruction fetching process loads two adjacent instructions at once from the program memory.

The PA is composed of two different kind of internal registers: Morphological Registers (MOR) and Logical Registers (LOR). The MOR consist of 5-bit south-to-north shift-registers whose southernmost position is loaded one image line at a time by the data bus. The presence of the 5-bit shift-registers allows to perform morphological operations on an enhanced 3×3 neighborhood, which is shown in Fig. 1(b). The LOR are used as a temporary storage and to perform logical operations.

2.2. Instruction set

In a generic Assembly instruction (called *morphological instruction*) two operations, a morphological one applied to a given MOR and a logical one applied to the previous result and to a second given register (MOR or LOR), can be performed in sequence. The result (1-bit) is placed in a third register:

```
R(25) = MATCH R(12) OR R(7)      ; A given match is performed on R(12);  the
                                 ; result is ORed with R(7) and stored in R(25)
```

The Assembly instructions are coded into fixed-length 32-bit words, where the first 5 bits are reserved for the operation code. Anyway, some instructions may need a number of operands whose total length exceeds 27 bits; this is the case of *morphological instructions* (where the complete matching template, the logical operation, the destination, and the two source registers must be specified) and *conditional jumps* (where both the condition and the new memory location must be expressed). These cases are handled by two separate instructions. In the former, a first TEMPLATE instruction is used to set the matching template [22,12] which is used by each following morphological operation. The template used in the following example corresponds to a 4-neighbors morphological erosion:

```
TEMPLATE–, –, 1, –, –, 1, 1, 1, –, –, 1, –, –     ; sets the 13 values corresponding to
                                                  ; the enhanced 3 × 3 matchingtemplate
R(18) = MATCH R(5)                                ; performs the match on register R(5)
```

In the second case, conditional jumps are split into two adjacent instructions: the former evaluates the condition and sets a specific flag. In case the flag is evaluated true, the latter is executed; otherwise the latter is treated as a NOP (No OPeration) instruction.

```
ONM > 5               ; evaluates the condition and sets the flag
  BRC < address >     ; jumps if the condition was evaluated true;
                      ; otherwise BRC is treated as a NOP instruction
```

This solution is particularly efficient for the conditional execution of any other instruction without using conditional jumps: for example,

```
ON M <= 7                            ; the morphological instruction is executed
  R(6) = MATCH R(5) AND R(8)         ; only when M <= 7
```

This approach allows to test easily also the combination of two or more conditions, as shown in the following example:

```
ON < Condition1 >       ; tests condition 1
  ON < Condition2 >     ; tests condition 2
    < INSTR >           ; execute instruction
```

Instruction `INSTR` is executed when both `Condition1` and `Condition2` are evaluated true and when `Condition1` is evaluated false, namely $(C_1 \cap C_2) \cup \overline{C}_1$.

Although loops can be implemented with conditional jumps, when the number of iterations is known at *assembly time*, the FOR-J construct can be used. [3] It has been introduced both to improve performance and to shorten the code.

```
FOR J = 0 TO 7 DO 2                ; Repeat the following < 2 > instructions with
  R(7) = R(7) OR R(9 + J)          ; counter J running from 0 to 7
  R(20 + J) = R(9 + J) AND R(3)    ;
```

PAPRICA-3 features many other Assembly instructions, ranging from the communication among PEs not directly connected [6] to the direct acquisition and output of images from a camera and to a monitor via a dedicated I/O bus. Nevertheless, the detailed description of these instructions is not presented here since it is not relevant to the following discussion and can be found in `http://www.CE.UniPR.IT/paprica3`.

3. The programming environment

The intrinsic complexity of Assembly language, especially when combined with the SIMD computational model, produces a low programming efficiency; a high level programming language, together with a software tool for the generation of its corresponding Assembly code, are then needed to ease the development of software applications. This section gives an overview of the approach, describes the classes available to the user and discusses the optimization problems that must be faced.

3.1. The `P3Lib` code generator

Two different solutions were considered for the generation of the Assembly code:

[3] A single counter "`J`" can be used, thus no nested constructs are allowed.

1. a *compiler*-based solution, where a new high level language is compiled and its corresponding PAPRICA-3 Assembly code is generated; and
2. a *code generator*-based solution, where an existing high level language is enhanced by a library of functions that parametrically build segments of PAPRICA-3 Assembly code.

The main disadvantage of the first solution lays in the need for a complete definition of a new language, which must be able to handle not only parallel statements, but also some other sequential functions (such as serial operations, data I/O, input of data from the user, memory management, etc.) which are of basic importance although not dealing with the specific SIMD extension. Conversely, a code generator inherits these capabilities from the chosen high level language and thus its implementation is considerably less complex than the former. Moreover, in this second case the efficiency in the development of applications is sensibly increased, since the programmer does not have to learn a new language but only its parallel extensions. Moreover, the use of an object oriented language eases the programmer task since the operands acting on the new objects can be used intuitively by the application programmer thanks to the possibility of overloading known operators.

These considerations led to the choice of a Code Generator based on the C++ object oriented language. The only drawbacks of this solution are (i) the low level of optimization achievable in a single pass (generally compilers and assemblers use two passes), and (ii) the impossibility to determine a mapping between a single C++ statement and its corresponding sequence of PAPRICA-3 Assembly instructions, thus not allowing the step-by-step execution of the C++ high level program for debugging purposes.

The complete sequence of operations required to build a PAPRICA-3 Assembly program is depicted in Fig. 2: first the C++ code is written and compiled; then the executable obtained is run and the PAPRICA-3 Assembly code is generated. Note that the generation of *Assembly code* was preferred with respect to the generation of *binary machine code*, thus requiring a further processing step (PAPRICA-3 Assembler) to get the final binary code. This choice allows a faster and more efficient debugging of the code-generation functions, as well as a simpler implementation of the following global optimization step, since in the Assembly code also comments are automatically included to ease the program comprehension. Moreover, since PAPRICA-3 has been conceived as a *real-time* image processor, a code optimizer can be used to furthermore improve the generated code. Finally the PAPRICA-3 Assembler is used to produce the binary machine code.

3.2. C++ classes

Two different C++ classes have been considered, corresponding to two different levels of abstraction:

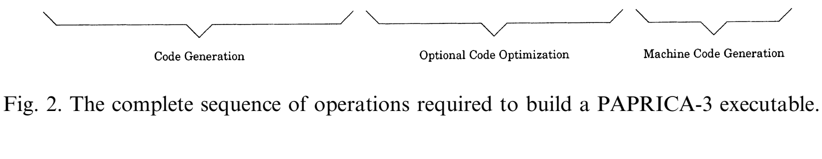

Fig. 2. The complete sequence of operations required to build a PAPRICA-3 executable.

- a *generic image*, thus not matching the system architecture (high level of abstraction like in the Khoros system [18]) and
- an *image line*, thus matching the system architecture (low level of abstraction like in the Connection Machine C* [15]).

In the former the user has no visibility at all over the hardware system and consequently the code generator must manage all data transfers and register allocation required by the virtualization of the PA (namely the iteration of the process over all the lines of the image), exploiting in the best way the architectural capabilities and hardware characteristics of the machine. Thus, the high level solution featuring extremely general image-wise primitives (such as gradient-based filters, convolutions, pattern-matchings, neural functions, etc.) offers a simple way of programming, but unfortunately the automatic generation of an optimized Assembly code is extremely complex due to the generality of the user primitives.

Conversely the low level solution matches the architecture of the system and gives to the user visibility over the hardware architecture, anyway hiding the details of processor virtualization. This last solution has been chosen for the PAPRICA-3 programming environment: the user can then focus on the application rather than on the implementation details.

3.2.1. The `P3_reg` class

The most important elementary class available to the user is the P3_reg class, representing *a set of binary internal registers* (Q-bit wide) of the PA. The following C++ program shows an example of high level programming of PAPRICA-3 using the P3_reg class. The following C++ code

```
P3_reg a(8);    //Definition of 3 registers with different depth: 8, 8,
P3_reg b(8);
P3_reg c(9);    //and 9 bits respectively
c = a + b;      //c is assigned the result of the sum between a and b
```

generates the following PAPRICA-3 Assembly code

```
; [39 - 47] = [56 - 63] + [48 - 55]          automatically generated comment
  R(39) = R(56) + R(48) + 0                  sum between less significant bits (no carry is used)
  FOR J = 0 TO 6 DO                          loop over the remaining 7 bits
    R(40 + J) = R(57 + J) + R(49 + J) + ACC  add with carry
  NEXT J
  R(47) = R(0) + NOT R(0) + NOT ACC          copies in R(47) the status of the carry flag
```

where register `a` is automatically allocated in registers `R(56)\XiR(63)`, `b` in `R(48)\XiR(55)`, and `c` in `R(39)÷R(47)`.

3.2.2. The `P3_mem` class

In the same way, the `P3_mem` class is defined representing a multi-bit *image* (Q-bit wide) in the Image Memory.

3.3. Code generation problems

PAPRICA-3 is a 3 address machine [16,17], namely in each Assembly instruction a maximum of 3 operands can be specified (two sources and one destination), as specified in Section 2.2. Thus for the complete code generation all 3 operands must be known. Unfortunately the C++ code generation approach [3] provides only two parameters to the member function in charge of the parametrical code generation and thus only a partial generation is possible. As an example, let us consider the two following C++ statements:

```
a = (b | c);
a = (b | c) & d;
```

In both cases the overloaded C++ "`operator |`" is the first function that is called: the two objects `b` and `c` are the only operands accessible by this function, and thus the *complete* code generation can be performed only when also the third object (the destination) is known. In the first case the destination object is `a`, while in the second case a new temporary object `tmp` must be allocated and used as destination. In fact, the second case can only be handled as two separate steps:

```
tmp = b | c;
a = tmp & d;
```

To overcome this ambiguity, the new class `P3_regTmp`, transparent to the user, has been introduced. It is composed of 3 fields (two `P3_reg` objects and an operand) and used when the complete code generation is impossible. The following example presents the sequence of functions called by the C++ compiler for the following C++ statement: `a = (b | c) & d;`

- `b | c`in this case (only two operands are specified) no generation is performed. A new `P3_regTmp` object (called `tmp1`), whose 3 fields are {`b; c; oper_or`}, is returned.
- `tmp1 & d`a new temporary `P3_reg` (called `t`) is allocated and the Assembly code corresponding to `t = tmp1` (namely `t = b | c`) is generated. The return value is a new `P3_regTmp` (called `tmp2`) whose 3 fields are {`t; d; oper_and`}.
- `a = tmp2`the Assembly code corresponding to `a = tmp2` (namely `a = t & d`) is generated and the temporary `P3_reg t` is deallocated.

3.3.1. Code optimization

Four different optimization levels have been considered: the first two can be handled by the code generator and are discussed in this section. conversely, the other two cannot be handled by the code generator because of it produce code in a single pass; thus they are performed by another tool (the Assembly code optimizer) and addressed in the next section.

1. *Local (or function-wise) optimization:* it is performed by each code generator function, which, instead of calling general-purpose procedures (in precompiled libraries), creates a segment of code optimized for the special set of parameters. Memory management and register allocation is performed at this stage.

2. *Look-behind (or non-backward) optimization:* in the generation of a sequence of instructions with no data-dependent points of divergence or merging of the program flow (hereinafter defined as a *block*), the knowledge of the code previously created can be exploited to drive specific optimizations. For example the

second of two morphological operations based on the same template will not generate a `TEMPLATE` instruction, since it has already been set by the former.

3. *Blocky (or block-wise) optimization:* it is performed by the stochastic code optimizer presented in Section 4.2, and it is required because of the lack of a second pass in the code generation. Optimizations like instruction scheduling, useless code removal, loop fusion and breaking, loop unrolling and rolling, etc. at the block level are performed at this stage through a stochastic approach, preceded by a deterministic preprocessing of the code.

4. *Global (or program-wise) optimization:* this kind of optimization requires the handling of more than one block at the same time; it is the only step in which data-dependent control flow constructs must be considered. In the literature [4] it is still addressed as an open problem currently under study.

Local optimizations: The first optimization level (*local* optimization) is performed during the code generation process: 3 operands are provided to each member function, as mentioned in Section 3.3, and the appropriate procedure is called to produce the corresponding Assembly code. A specific set of parameters selected from the 3 operands (the 3 `P3_reg` objects) is passed to the code generation function and used to drive specific optimizations. As an example, let us consider the following C++ statement:

```
a = (b & c) | d;
```

where `a`, `c`, and `d` are 8-bit deep `P3_reg` objects, while `b` is only 4-bit deep. The depth (number of bits) of the temporary `P3_reg` object `tmp` used in `tmp = b & c` (see Section 3.3) depends on the depth of the two operands (`b` and `c`) and on the specific function performed (*logical and*). In this case a 4-bit temporary `P3_reg` is sufficient since `b` (which is 4-bit deep) is considered padded with zeros and thus the number of significative bits of the result of a *logical and* between `b` and any other deeper `P3_reg` is 4. Assuming `a` allocated in `R(56)÷R(63)`, `b` in `R(52)÷R(55)`, `c` in `R(44)÷R(51)`, and `d` in `R(36)÷R(43)`, the corresponding Assembly code automatically generated is in fact:

```
Allocating registers [32 – 35]          Allocating the temporary P3_reg tmp (4-bit deep)
; [32 – 35] = [52 – 55] & [44 – 51]     Performs a logical AND between b and the 4
  FOR J = 0 TO 3 DO                       less significant bits of c and   stores the
    R(32 + J) = R(52 + J) AND R(44 + J) result into the 4-bit deep temporary P3_reg
  NEXT J
; [56 – 63] = [32 – 35] | [36 – 43]     Performs a logical OR between the previous
  FOR J = 0 TO 3 DO                     result and the less significant bits of d
    R(56 + J) = R(32 + J) OR R(36 + J)
  NEXT J
  FORJ = 0 TO 3 DO                      Copies the 4 more significant bits of d in the
    R(60 + J) = R(40 + J)               4 more significant bits of the result a
  NEXT J
; Deallocating registers[32 – 35]
```

This corresponds to the execution of 4 logical AND, 4 logical OR, and 4 assignments. On the other hand, this optimization cannot be exploited in the following case:

```
a = (~b & c) | d;
```

where a bitwise inversion causes b to be considered padded with ones. The code generated in this case is in fact:

```
; Allocating registers [28 − 35]             Allocating the temporary P3_reg tmp (8-bit deep)
; [28 − 35] =~[52 − 55] & [44 − 51]          Performs a logical AND between ~b and the
  FOR J = 0 TO 3 DO                            4 less significant bits of c and stores the
    R(32 + J) = NOTR(52 + J) AND R(44 + J)     result into the 4 less significant bits of the
  NEXT J                                       temporarty P3_reg
  FOR J = 0 TO 3 DO                          Copies the 4 more significant bits of c in the
    R(32 + J) = R(48 + J)                      more significant bits of the temporary P3_reg
  NEXT J
; [56 − 63] = [28 − 35] | [36 − 43]          performs a logical OR between the temporary
  FOR J = 0 TO 7 DO                            P3_reg and d
    R(56 + J) = R(28 + J) or R(36 + J)
  NEXT J
Deallocating registers [28 − 35]
```

This corresponds to the execution of 4 logical AND, 8 logical OR, and 4 assignments.

Look-behind optimizations: Since the generation process implemented here can only *add* Assembly lines to the final PAPRICA-3 code (no backward generation or editing is in fact possible), the code segments which do not include control flow statements (FOR-NEXT, REPEAT-UNTIL, IF-ELSE-ENDIF constructs) can be furthermore optimized.

As an example, let us consider two adjacent morphological operations using the same matching template: the following C++ code

```
P3_reg a(8), b(8);             // Allocates two P3_reg objects
a = a.match(P3_NMOV);          // Shifts a to the North
b = b.match(P3_NMOV);          // Shifts b to the North
```

generates two loops (corresponding to the two morphological operations), but only *one* TEMPLATE instruction is generated since both operations make use of the same matching template:

```
; Allocating registers [56 − 63]             allocates registers for object a
; Allocating registers [48 − 55]             allocates registers for object b
; [56 − 63] = [56 − 63], match = P3_NMOV     corresponds to a = a. match(P3_NMOV)
  TEMPLATE = −, −, −, −, −, −, −, −, −, −, 1, −, −
  FOR J = 0 TO 7 DO
    R(56 + J) = MATCH R(56 + J)
  NEXT J
```

```
; [48 - 55] = [48 - 55], match = P3_NMOV      corresponds to b = b. match(P3_NMOV)
  FOR J = 0 TO 7 DO
    R(48 + J) = MATCH R(48 + J)
  NEXT J
; Deallocating registers [48 - 55]            deallocates b
; Deallocating registers [56 - 63]            deallocates a
```

In other words, this optimization aims to remove all the instructions that do not modify the status of the Finite State Machine implementing the PA.

4. Assembly code optimization

Since the code generator produces the assembly code in a single pass, a backward optimization is not possible. Thus a tool for the optimization of the assembly code has been developed.

The efficient use of a pipelined processor is mainly based on a specific ordering of the program instructions (known as *instruction scheduling*), aimed to the maximization of the number of the elementary operations executed simultaneously by the different pipeline stages. A lot of different approaches have been conceived and implemented to solve the *instruction scheduling* problem: although in Ref. [10] it is demonstrated that instruction scheduling is a NP-complete problem, some polynomial solutions have been proposed, such as *list scheduling* [2,4]. But unfortunately, deterministic solutions become even more complex when, in addition to instruction scheduling, other optimization criteria are combined.

Conversely, in the development of the PAPRICA-3 environment a stochastic approach has been followed for the complete optimization (instruction scheduling, instruction alignment, loop fusion and breaking, loop unrolling and rolling, etc.) of Assembly code. It is based on an evolutionary technique in which genetic rules iteratively modify a population of programs. The set of rules include both well-known traditional optimization criteria [1] (instruction scheduling, useless code removal, loop fusion and breaking, loop unrolling and rolling, etc.) and rules specifically designed to optimize the Assembly language of the PAPRICA-3 system.

The optimization process is divided into two main steps: a deterministic one, performed on the initial assembly code, and a stochastic one, iterated on the result of the previous step. The first step is fully deterministic and reduces both execution time and code size; on the other hand, the stochastic step tends to improve the efficiency of the Assembly code exploiting the characteristics of the pipeline structure, the specific instruction set encoding, and other hardware features. The stochastic process is driven by the user choice about the optimization target: execution time, code size, or a given combination of both.

4.1. Deterministic optimization

As described above, PAPRICA-3 assembly code is produced by a "C++ to Assembly" code generator. Since C++ statements are consecutively and independently processed, code generation is performed in a single pass, namely the optimization during code generation can only exploit the knowledge of the already created code. For this reason the generated Assembly code features some redundancies, whose removal is performed by a deterministic optimization step. The techniques used are discussed in the following.

- *Invariant code motion:* The single pass-based generation does not allow to know whether the body of a loop will contain invariant code prior to the complete generation of the loop itself. As a result, the loop body may comprise some instructions which are independent of the loop index and independent of the other loop instructions. These instructions can be pulled out of the loop body, thus reducing the execution time.
- *Removal of Jumps:* Conditions expressed in C++ are rendered by the code generator using a sequence of `ON` and `BRC` instructions. In case the statement executed when the condition is evaluated true is composed of a single assembly instruction (see Section 2.2), the `BRC` and its corresponding label can be removed and the condition in the `ON` instruction complemented, thus reducing both execution time and code size. Nevertheless, the most important benefit is the removal of a label and a branch, which identify points of flow merging and divergence respectively. In this way the number of blocks is reduced, thus helping the following optimization.

4.2. Stochastic optimization

Since in the optimization phase no data are available, data-dependent branches cannot be solved. For this reason, the code is divided into *blocks*, which are defined as code segments without data-dependent conditional control-flow constructs. Each block is optimized independently of the others: a first module splits the code into blocks, and sends them to independent processes in charged of the optimization, as shown in Fig. 3. Finally the optimized blocks are re-assembled and the final program is generated.

Each block is optimized following an iterative stochastic approach: at the *i*th iteration, N(*i*) new programs are generated by the consecutive application of R(*i*) rules (drawn randomly from a given set) to each program of the (*i*-1)th generation. The ones performing better than a given threshold *T*(*i*) are kept and will form the *i*th generation, while the other are discarded. The timing performance of each new program is determined using the PiPE tool (see Section 5).

Since the population at the first iteration is composed by only the original block, at the beginning of the process (when *i* is small) the number of rules applied, R(*i*), is high, thus allowing a large spreading of the

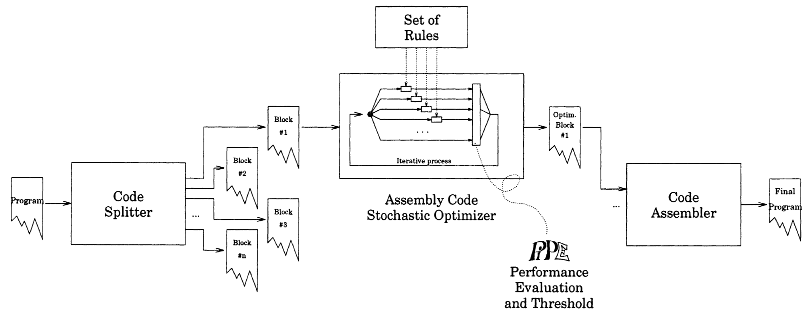

Fig. 3. Block diagram of the stochastic code optimization.

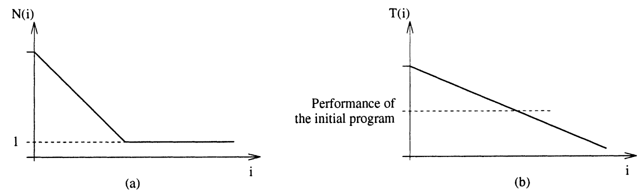

Fig. 4. Qualitative behavior of (a) the number of rules applied consecutively to a single block; (b) the threshold $T(i)$ as a function of the iteration number i.

initial population. As soon as i becomes large, the number of rules applied to the same block decreases [4] to 1, for a fine tuning of the solution (see Fig. 4(a).

Fig. 4(b) shows a qualitative behavior of threshold $T(i)$, which, starting from a value twice as large as the performance index featured by the initial program, decreases linearly [5] with i. This implies that at the beginning of the iterative process (i small), solutions with performance lower than the initial can be kept (this is of basic importance to escape from local minima), while in the following iterations a threshold limits the number of candidates and keeps only the best solutions. At each iteration the block used to generate the new population is also included in the evaluation process, in order to avoid jittering from the optimal solution that may have been eventually detected. The number of iterations of the stochastic process and the slope of $T(i)$ is a function of the length of the block to be optimized: in the first version its dependence is linear.

Each rule is drawn randomly, according to a given probability. In the current version, the probability is fixed, but in future versions it will be modified run-time to allow a faster convergence to the optimal solution. Each rule can move, modify, add, or remove instructions. Every action is fully reversible, thus allowing to return to the originating state. The list of elementary rules is described in the following.

- *Instruction scheduling:* For this purpose the concept of *independence* between instructions is used. Two instructions are said to be *independent* when their swapping does not alter the result of program execution for every input. This happens both when the data modified by the first instruction are not used neither by the latter nor by the instructions between the two, and when the data used as source in the former are not modified neither by the latter nor by any of the instructions in between. Obviously, the data modified or used by a general instruction are both registers directly specified in the opcode and data intrinsically involved in the processing (such as flags and the accumulator). The application of this technique is based on the swapping of two independent instructions, and produces new versions of the code leading to a different execution time without changing the code length. This rule is auto-reversible.
- *Instruction alignment:* In this case the optimization can take advantage from the specific fetch policy of PAPRICA-3. In order to increment the working frequency of the processor, bounded by the memory cycle time, PAPRICA-3 fetches two instructions (placed at an even and odd adjacent memory locations)

[4] The linear slope of R(i) (resembling a *simulated annealing* approach) was chosen to simplify the development of the tool, but more complex functions are under evaluation.

[5] The slope of $T(i)$ determines the convergence speed.

every two processor cycles. Unfortunately when a branch to an odd address occurs, PAPRICA-3 must fetch and ignore the instruction at the previous even address. The addition (or removal) of an instruction (i.e. a `NOP`) can transform an odd address into an even one and vice versa.

- *Loop fusion and breaking:* Two different adjacent `FOR` cycles can be joined when their counter `J` ranges into the same interval and all the instructions of one loop body are independent of all instruction of the other loop body for each value of `J`. The dual operation, making the rule reversible, is based on the creation of two disjoint loops when both the body of the original one has more than one instruction and all the requirements on independence described above are met.
- *Loop unrolling and rolling:* In pipelined processor execution time is bounded especially by control flow instructions [13]. In order to fully exploit the instruction scheduling and to avoid control hazards, loops can be unrolled. Unfortunately, the main drawback of the loop unrolling technique is the significant increment in code size. The final measurement of the performance of a given block is performed as a weighted average between the number of clock cycles and the length of the code, according to the user choice (speed *vs*. code size, or a given mix of the two).
- *Instruction fusion and breaking:* PAPRICA-3 morphological instructions are a concatenation of a morphological operator and a logical one. Therefore such instructions can be split in two adjacent ones, performing the two operations independently. This optimization is obviously specific to the PAPRICA-3 system. Although this operation increases code size, two instructions that use different source data can take advantage from the instruction scheduling technique. Since this rule is reversible, two instruction can be fused, thus decreasing code size and, in case of no data hazards, also execution time.
- *Multiple assignment removal:* When two adjacent instructions perform an assignment to the same destination (i.e. the same internal register, the same memory location, etc.) and the latter does not involve the destination as source, the former is useless and can be removed. This condition can be produced by the application of other rules, such as the *instruction breaking or fusion.* The dual operation, making this rule reversible, is based on the insertion of a generic assignment that has the same destination as the following instruction.
- *Removal of temporary registers:* The code generator allocates temporary registers for storing intermediate results; moreover programmers themselves do the same with temporary variables. Due to the possibility to execute a morphological and a logical operation in the same PAPRICA-3 instruction, there are cases in which there is no need to use temporary registers. In the following example `R(6)` is a temporary register: the first instruction could be removed only if `R(6)` is not used in the following part of the program (i.e., if `R(6)` is a temporary register).

```
First case                 Second case
R(6) = NOT R(7)            R(6) = NOT R(7)
R(5) = MATCHR(8) & R(6)    R(5) = MATCHR(8) & NOTR(7)
```

The reversible rule is anyway straightforward. Unfortunately, at assembly level, it is difficult to de- termine the scope of a register; moreover the choice of optimizing the code block by block makes it impracticable. Thus it is impossible to remove the assignment to `R(6)` since it is not known if its value will be used again. Conversely PAPRICA-3 code generator knows the scope of every register and can generate the *dummy* instruction FREE that signals when the scope of a register is over. Obviously the FREE instruction is ignored both during the evaluation phase and by the compiler, thus not affecting both performance evaluation and program execution; FREE R(6) is treated as an assignment to register R(6). The use of the dummy instruction FREE together with the *multiple assignment removal* and *instruction scheduling* rules allows to improve sensibly the code, as in

```
First case                   Second case
R(6) = NOT R(7)               R(5) = MATCH R(8) & NOT R(7)
R(5) = MATCH R(8) & R(6)     FREE R(6)
FREE R(6)
```

- *Registers Reallocation:* The policy of register allocation used by the PAPRICA-3 code generator is aimed to minimize data hazards: the registers allocated are chosen among the available ones that has been idle for the longest period of time. Unfortunately, the random application of optimization rules (especially *instruction scheduling, instruction breaking, or removal of temporary registers*) can affect the idle time of the registers, thus modifying the compile-time situation. This rule provides a way of reallocating temporary registers to decrease data hazards, provided that another register is available for temporary storage. A register can be reallocated as in the following example.

```
First case                   Second case
FREE R(15)                   FREE R(15)
...                          ...
R(6) = NOT R(7)              R(15) = NOT R(7)
R(5) = MATCH R(8) & R(6)     R(5) = MATCH R(8) & R(15)
FREE R(6)                    FREE R(15)
```

5. Performance evaluation

Due to the complex internal structure of each PE, the determination of the execution time of a given program is unpredictable [13,14] without a simulation or a real run. Since the hardware system is not yet available, a tool for performance evaluation of a *general* pipelined processor was developed: PiPE, Pipeline Performance Evaluator. The development of a simulator of a *general* pipelined processor allowed to dynamically modify the internal architecture of the system and to test and evaluate different possible solutions. Success in developing such a tool is tightly coupled to its flexibility and its effectiveness in modeling different pipelined architectures, since a generic pipeline can be composed of any number of stages, each one with different functionalities.

The PiPE tool allows to model different behaviors, such as:
- different functional units;
- multiple functional units of the same type;
- data, structural, and control hazards;

- multiple or conditional data paths;
- interrupts or external signals.

The customization of the PiPE tool for the description of a specific processor involves the definition of the following informations:

- how the instructions are coded;
- a description of the pipeline topology;
- the data-path followed by each instruction;
- hazard conditions;
- all the specific I/O mechanism.

Moreover since the target is the development of a *fully general* performance evaluation tool based only on a behavioral simulation (i.e. not a *functional* simulator), some implementation-specific problems and data-dependent events must be faced; for example the handling of:

- different instruction fetching methods (multiple fetches as in super scalar architectures),
- registers values and counters,
- external synchronizations and interrupts,
- evaluation of conditions.

6. Implementation

This section describes the implementation issues that have been addressed in the development of the whole environment.

For portability purposes the entire code has been written using standard programming languages and tools, such as `ANSI C`, `C++`, `LEX`, `YACC`, and `MPI` libraries, and runs both on `UNIX` based machines (SunOS and Linux) and in `DOS` based environments (MsDos and Windows).

6.1. The PiPE software simulator

Currently the whole PAPRICA-3 system is simulated on serial machines. The software simulation allowed to test the effectiveness of the instruction set, the choice of the PE neighborhood, the logical organization of the Image Memory, the internal structure of the processor, as well as the performance of the whole system prior to the hardware availability. In fact the implementation and simulation of some basic image processing functions led to some architectural improvements, mainly focused on the tuning of the instruction set and on the redesign of the pipeline topology.

Starting from the experience gained in the development of the first PAPRICA prototype [8], two different simulators were originally planned:

- a *system-level* simulator, capable of emulating the hardware behavior, thus allowing the exact determination of the number of clock cycles and memory accesses required by the application; and
- a *functional* simulator, aimed to the simulation of the algorithms rather than the hardware system, starting directly from the C++ source.

These two simulators had to interface directly to the C++ application, giving to the programmer the possibility to choose the desired simulation engine: the functional simulator, for the development and debugging of the application, while the system-level simulator, for the its final tuning and performance evaluation.

6.1.1. The System-level simulator

Due to the simple operations of PAPRICA-3 instruction set (single-bit operations), an extremely efficient simulation can be obtained by exploiting the word parallelism ω of the single processor of the serial system running the simulation [23]. The Q bits of the PA registers (see Section 2.1) are grouped into $\lceil Q/\omega \rceil$ clusters of ω bits each, and processed *simultaneously* by the serial processor. In the current implementation it is $Q = 64$ and $\omega = 32$ and thus 2 ($= 64/32$) cycles are required to accomplish the whole processing of a single binary image line. This leads to a sort of parallelization in the software simulator based on a general-purpose serial processor. A real parallel version was considered but not developed, due both to the extremely high communication requirements and to the satisfying performance of the current serial implementation: the ratio between the simulation time and the processing time computed by PiPE ranges from 450:1 (on a SUN Sparc20 with a 50 MHz SuperSPARC) to 300:1 (on a PC equipped with a 100 MHz Intel Pentium).

6.1.2. The functional simulator

Beside the system-level simulator described above, also a functional simulator was planned, but never developed. Some simple considerations on computational efficiency in fact showed that the functional simulator would perform worse than the system-level simulator. As an example, let us consider the simulation of the following common operation:

```
a = b | c;
```

namely a logical operation between two 8-bit deep (64 bit wide) image lines on a serial processor with 32-bit word parallelism. The system-level simulator requires 16 (corresponding to $\frac{64}{32}$ cycles of 8 operations each) 32-bit operations, while the functional simulator requires 64 (one for each pixel) 8-bit logical operations. The high hardware efficiency (which corresponds to a high computational efficiency) obtained in the first case suggested the development of the system-level simulator only.

6.2. The PaPRICA-3 tool

The PiPE system has been developed in C++ because an object-oriented language allows to hide implementation details behind different classes, which can be used to model different pipeline stages.

Any pipeline can be completely defined with the description of:
- the list of stages composing the system;
- the interconnection topology which links the stages;
- and the flow of instructions between the stages.

In the PiPE tool the hardware details are encoded in two plain ASCII files which describe the routing of each instruction into the pipeline, and the instructions encoding; from these information also the pipeline topology and hazards conditions are deduced. The possibility to configure the PiPE tool acting only on external customization files allows a fine and efficient tuning of the hardware system according to a standard hardware-software codesign methodology.

6.3. The code optimizer

Since the major drawback of a stochastic approach to code optimization is the high computational power (in terms of both memory usage and CPU time), a parallel version of the optimizer has been developed. The development of the parallel version started with the analysis of the system requirements and performance on a preliminary serial version of the optimizer:

- the CPU time spent in the performance evaluation phase by the PiPE tool is up to 90% of the overall CPU time;
- the larger the number of iterations, the better the quality of the result, but also the larger the amount of memory required.

Fig. 5 shows that the complete process involves the repeated execution of some different steps, which can be distributed among different processors, leading to a different granularity of the distribution. Different approaches for the development of the parallel version of the optimizer were considered, each based on the distribution of different processing steps: the whole optimization of each single block, the genetic process, the performance evaluation step, or a given mix of them. The elementary processing steps and their requirements are summarized in Table 1; code splitting and assembling are anyway serial steps which are performed by a single *master* processor.

Fig. 6 shows three different solutions to the parallelization problem, whose characteristics are de- scribed in Table 2.

The approach followed in the development of the first distributed version is based on the distribution of the performance evaluation step only (solution (c)): a master processor splits the code into blocks and generates the new blocks to be evaluated. The new blocks are queued and sent to different processors for the evaluation of their performance. When a processor completes the evaluation phase, it returns the number of clock cycles to the master node and signals its availability to evaluate a new code segment. Due to the

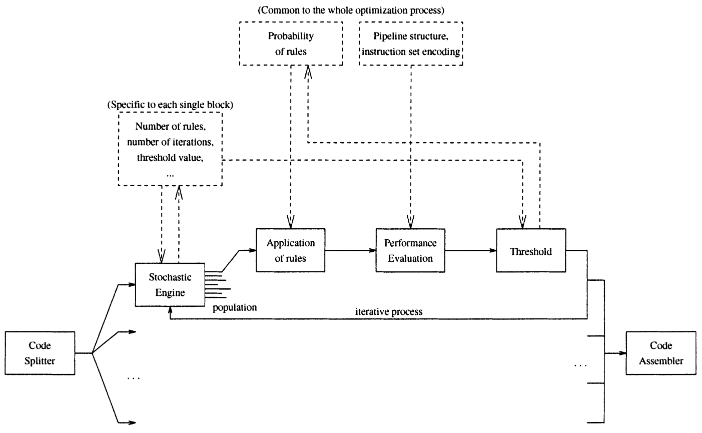
(Common to the whole optimization process)
Probability of rules
Pipeline structure, instruction set encoding
(Specific to each single block)
Number of rules, number of iterations, threshold value, ...
Application of rules
Performance Evaluation
Threshold
Stochastic Engine
population
iterative process
Code Splitter
Code Assembler

Fig. 5. Different steps of the processing.

Table 1
The elementary processing steps and their requirements

	Code splitter	Stochastic engine	Application of rules	Performance evaluation	Threshold	Code assembler
CPU usage(%)	Negligible	6	3	90	1	Negligible
Requirements	Executed only once for each run	Requires a lot of memory to handle large populations; updates the threshold, number of rules, number of iterations,...	Reads the probabilities of each rule	Reads the pipeline structure and instruction set encoding (fixed for each run)	Reads the threshold value (changes at each iteration) and modifies the rules probabilities	Executed only once for each run

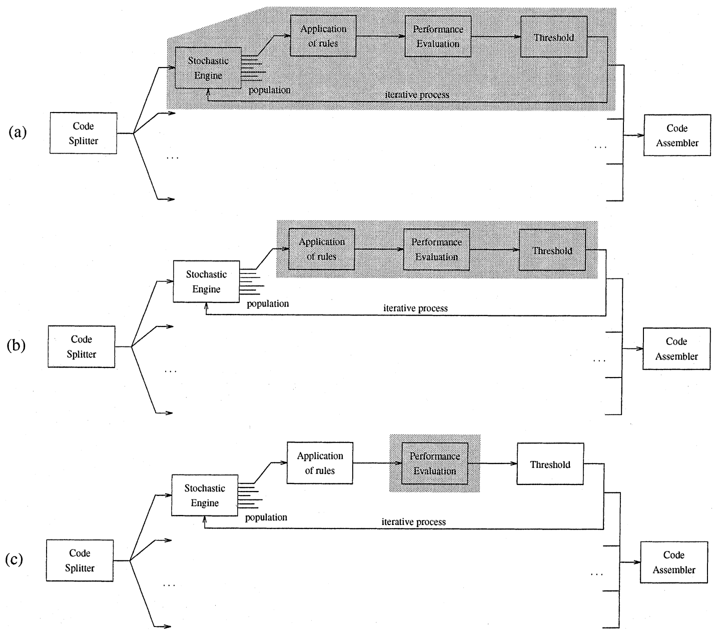

Fig. 6. Three different solutions to the parallelization problem: the gray areas represent the portion of the processing which is assigned to a single processor of the cluster; the remaining white blocks are assigned to the *master* processor.

different computational power required by (i) the evaluation phase (90%) and (ii) the rest of the processing (10%), this approach balances the load even in the case of a non-homogeneous cluster of processing nodes.

The rules that are currently operative in the first version of the tool and that have been used in the example described in the following section are: Invariant Code Motion, Removal of Jumps, Instruction

Table 2
Characteristics of the three solutions described in Fig. 6

Parallelization	Load balancing	Communications	Memory usage	busy time/transfer time
(a) Each single block is optimized independently of the others; each block is assigned to a single processor	Load balancing cannot be guaranteed: it depends on the number and length of the blocks in a program	Very low: only the original and optimized blocks are transferred	Normal; well balanced	Extremely high
(b) Each processor applies a set of rules to each block it receives, evaluates its performance, and compares it with the threshold	Well balanced: as soon as a processor finishes its task, it receives a new block from the master processor	High: besides the original and optimized blocks each processor reads and modifies a set of parameters allocated in the master node	Low amount for each node, but high amount for the master node, which has to manage the whole population	High
(c) Each processor receives a block and evaluates its performance	Well balanced: as soon as a processor finishes its task, it receives a new block from the master processor	High, but better than in the previous case since each processor receives a block and outputs a single number	Low amount for each node, but high amount for the master node, which has to manage the whole population	High, but lower than in the previous case

Scheduling, Instruction Alignment, Loop Fusion and Breaking, Loop Unrolling and Rolling, while Instruction Fusion and Breaking, Multiple Assignment Removal, Removal of Temporary Registers, Registers Reallocation, are currently under test. The future integration of these rules will hopefully lead to a higher degree of optimization.

7. An example

A number of different programs has been developed, tested, and optimized using the PAPRICA-3 operating environment; nevertheless, hereinafter we will refer to a specific example: an image processing algorithm for the detection of road markings in images acquired from a moving vehicle [5].

In this case the execution time plays a basic role, and thus code optimization becomes mandatory. The algorithm is based on a morphological filter which enhances the road markings, and on an adaptive threshold which discards pixels not belonging to a continuous line. The algorithm was written in C++ using the classes offered by the code generator; in this way the spatial parallelism intrinsic to the SIMD implementation of the algorithm and the virtualization of the processor array are completely handled by the code generator, leaving to the programmer only the task of specifying the vertical size of the image.

The C++ source code consists of about 90 lines, which, thanks to the code generator, are automatically converted into about 970 Assembly instructions. This non-optimized program runs in 2398 clock cycles, while after the optimization the new Assembly code runs in 1961 clock cycles. The improvement is about 18%. The time required for the optimization of this code on a cluster of 4 Pentium 100 with 32 Mbytes of RAM running Linux is about 25 min.

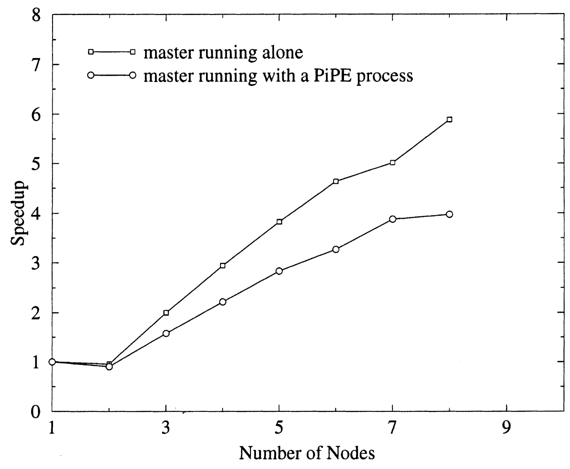

Fig. 7. Speedup as a function of the number of nodes.

The Code Generator and PiPE demonstrated to be effective and underwent a number of improvements which led to the current stable version. On the contrary, although the current version of the Code Optimizer produces satisfactory results, a number of improvements are planned, ranging from the introduction of new optimization rules to the study of new solutions for a different parallelization of the tool. For example the timing performance of the optimization has been evaluated on a different number of computing nodes in two different configurations: (a) one processor is exclusively allocated to the master process, while the others perform the evaluation step; (b) all processors perform the evaluation step, while one of them handles also the master process. Fig. 7 shows an average behavior of the Code Optimizer in these two situations. In the second case the performance is lower than in the former because of the high number of context-switching required by the simultaneous execution of two processes, one of which (the master) is extremely memory-consuming.

Table 3 shows a segment of the original code, its optimized version, and the resources used by each single instruction, where 'R', 'W', and 'RW' mean that the resource is accessed for reading, writing, or both respectively. The analysis of this example shows that the deterministic optimization moved the TEMPLATE instruction out of the block and that the two instructions referring to I1 and I2 have been swapped with their previous and following ones respectively. This simple rescheduling of the instructions allows to eliminate consecutive accesses (in 'W' and 'R' respectively) to the same resource, thus decreasing hazards.

8. Conclusions

In this paper the operating environment of the PAPRICA-3 system has been described.

The programming environment, based on C++, has proven to be very effective, since the user does not have to learn a new language, but programming becomes an easy task thanks to the use of overloaded

Table 3
Analysis of the result of the optimization of a short segement of code

	I1	I2	M	R(15)	R(60)	Templ.	Memory
Original Code LM00000L:							
LD I1 (0), R (15)	R			R			R
TEMPLATE = -,-,-,-,1,-,-,-,-,-,-,-,-						W	
R (60) = MATCH R (15) EXOR R (15)				R	W	R	
ST I2 (0), R (60)		R			R		W
ADD I1, A	RW						
ADD I2, A		RW					
INC M			RW				
ON M <= 31			R				
BRC LM00000L							
Optimized Code							
TEMPLATE = -,-,-,-,1,-,-,-,-,-,-,-,-						W	
LM00000L:							
LD I1 (0), R (15)	R			R			R
R (60) = MATCH R (15) EXOR R (15)				R	W	R	
ADD I1, A	RW						
ST I2 (0), R(60)		R			R		W
INC M			RW				
ADD I2, A		RW					
ON M <= 31			R				
BRC LM00000L							

operators acting on new C++ classes. Moreover since the system-level simulator delivers remarkable performance, an extremely efficient prototyping, testing, and tuning of applications becomes possible. On the other hand the tuning of the hardware system can be efficiently performed acting only on the configuration of the PiPE tool. Finally the use of The Code Optimizer allows to reach an excellent performance improvement (up to 35%) of the code previously generated, but, due to its high computational load it is particularly suited for the final optimization of programs whose computing time is a key parameter (i.e. real-time applications). The problem of the slowness of the stochastic process has been resolved with a parallel implementation of the optimizer on a cluster of PCs connected by Fast Ethernet; the speedup demonstrated to be almost linear with the number of computing nodes; nevertheless further experiments are now planned on a larger number of nodes. Although the current version of the Code Optimizer produces satisfactory results, a number of improvements are planned, ranging from the introduction of new optimization rules to the study of new solutions for its parallelization.

Acknowledgements

This work was partially supported by the Italian CNR under the frame of the Progetto Finalizzato Trasporti 2.

References

[1] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, Addison–Wesley, Reading, MA, 1986.

[2] S.J. Beaty, S. Colcord, P.H. Sweany. Using genetic algorithms to fine-tune instruction-scheduling heuristics, in: Proceedings of IEEE International Conference on Massively Parallel Computing Systems(MPCS-96), IEEE Computer Society – EuroMicro, 6–9 May 1996.

[3] G. Booch, Object-Oriented Analysis and Design-with Applications, Benjamin/Cummings, Menlo Park, CA, 1996.

[4] M.J. Bourke, III, P.H. Sweany, S.J. Beaty, Extending list scheduling to consider extension frequency, in: Proceedings of the 29th Hawaii International Conference on System Sciences, IEEE Computer Society, 1996, pp. 193–202.

[5] A. Broggi, The evolution of a massively parallel vision system for real-time automotive image processing, in: Proceedings of IEEE International Parallel Processing Symposium (IPPS'96), IEEE Computer Society, Honolulu, HI, 1996, pp. 724–728.

[6] A. Broggi, Global communications on a linear array architecture, Journal of Parallel Algorithms and Applications 11 (1/2) (1997) 27–43.

[7] A. Broggi, G. Conte, G. Burzio, L. Lavagno, F. Gregoretti, C. Sansoè, L.M. Reyneri, PAPRICA-3: A real-time morphological image processor, in: Proceedings of the First IEEE International Conference on Image Processing (ICIP), IEEE Computer Society, Austin, TX, 13–16 November 1994, pp. 654–658.

[8] A. Broggi, G. Conte, F. Gregoretti, C. Sansoè, L.M. Reyneri, The evolution of the PAPRICA system (Special Issue on Massively Parallel Computing), Integrated Computer-Aided Engineering Journal 4 (2) (1997) 114–136.

[9] A. Broggi, F. Gregoretti, in: A. Broggi, F. Gregoretti (Eds.), Editorial: Special-Issue on Special-Purpose Architectures for Real-Time Imaging, Real-Time Imaging Journal 2 (6) (1996) 329–330.

[10] D.J. DeWitt, A machine independent approach to the production of optimal horizontal microcode, Ph.D. Thesis, Dept. of Computer and Communication Sciences, University of Michigan, Ann Arbor (1976).

[11] F. Gregoretti, F. Intini, L. Lavagno, R. Passerone, L.M. Reyneri, Design and implementation of the control structure of the PAPRICA-3 processor, in: Proceedings of the Fourth EuroMicro Workshop on Parallel and Distributed Processing, IEEE Computer Society – EuroMicro, Braga, Portugal, 24–26 January 1996, in press.

[12] R.M. Haralick, S.R. Sternberg, X. Zhuang, Image analysis using mathematical morphology, IEEE Transactions on Pattern Analysis and Machine Intelligence 9 (4) (1987) 532–550.

[13] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, Morgan Kaufmann, Los Altos, CA, 1990.

[14] J.L. Hennessy, D.A. Patterson, Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann, Los Altos, CA, 1994.

[15] W.D. Hillis, The Connection Machine, MIT Press, Cambridge, MA., 1985.

[16] K. Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill, New York, 1993.

[17] J.P. Hayes, Computer Architecture and Organization, McGraw-Hill, New York, 1988.

[18] K. Konstantinides, J. Rasure, The Khoros software development environment for image and signal processing, IEEE Journal of Image Processing (1993).

[19] MPI Forum, MPI A Message Passing Interface Standard, Technical Report, University of Tennessee (1995).

[20] A. Nicolau, J. Fisher, Measuring the parallelism available for very long instruction word architectures, IEEE Transactions on Computers 33 (11) (1984) 968–976.

[21] B. Ramakrishna Rau, J.A. Fisher, Instruction level parallel processing: History, overview, and perspective, The Journal of Supercomputing (1993) 9–50.

[22] J. Serra, Image Analysis and Mathematical Morphology, Academic Press, London, 1982.

[23] M.R. Spieth, J.P. Hulskamp, Parallel image processing on single processor systems, in Proceedings of the Fourth IEEE International Conference on Image Processing (ICIP), vol. 2, IEEE Signal Processing Society, Lausanne, Switzerland, 16–19 September 1996, pp. 133–136.

[24] D.W. Wall, Limits of instruction level parallelism, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, Santa Clara, CA, 1991, pp. i176–188.

Alberto Broggi is Assistant Professor at the Dipartimento di Ingegneria dell'Informazione of the University of Parma since 1994. He received both the Dr.Ing. (Master) degree in Electronic Engineering (1990) and the PhD degree in Information Technology (1994) from the same University. His research interests include real-time computer vision algorithms for the navigation of unmanned vehicles, and the development of low-cost computer systems to be used on autonomous robots. He is the coordinator of the ARGO project, aimed to the design, development and test of the ARGO autonomous prototype vehicle of the University of Parma. He is the author of more than 100 refereed publications in international Journals, book chapters, and conference proceedings. He is the Editor of the Newsletter and member of the Executive Committee of the IEEE Technical Committee on Complexity in Computing, and member of the Editorial Board of Real-Time Imaging Journal, and Engineering Applications of Artificial Intelligence Journal. He has guest edited a number of special-issues of international journals on Machine Vision (IEEE Intelligent Systems, Image and Vision Computing, Real-Time Imaging, Engineering Applications of Artificial Intelligence) and has been invited to organize several minitracks and special sessions in international conferences (IEEE Intl Conf on Intelligent Transportation Systems, IEEE Symp on Intelligent Vehicles, IEEE Intl Conf on Algorithms And Architectures for Parallel Processing, SPIE Aerosense, Intl Symp on Automotive Technology and Automation, Hawaii Intl Conf on System Sciences). He served on the Program Committee and as Publicity Chair and Tutorial Chair of many major conferences.

Massimo Bertozzi was born in Parma in 1966. In 1994 he received the Dr.Ing.~degree in Electronic Engineering from the University of Parma discussing a Master Thesis about the implementation of Simulation of Petri Nets on the CM-2 Massive Parallel Architecture. From November 1994 to October 1997 he was a PhD student in information technology atthe Dipartimento di Ingegneria dell'Informazione of the University of Parma where he chaired the local IEEE student branch. During thisperiod his research interests focused mainly on the application ofimage processing to real-time systems and to vehicle guidance, on theoptimization of machine code at assembly level, and on parallel and distributed computing. Since November 1997 he has held a permanent position at the Dipartimento di Ingegneria dell'Informazione.